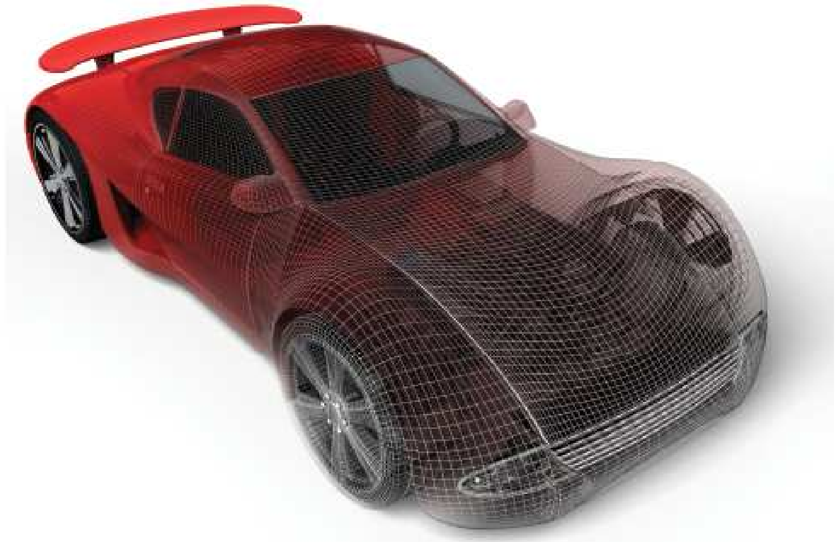
Test-Drive ASP.NET MVC

The Pragmatic Programmers

Covers ASP.NET MVC 2.0

Test-Drive ASP.NET MVC



Jonathan McCracken

Edited by Susannah Davidson Pfalzer

Great things are not done by impulse but by a series of small things brought together.

► Vincent van Gogh

Chapter 9

Integrating Repositories with Controllers

In the previous chapter, we built our first repository to access and modify the database. A repository on its own won't do anything unless it's tied back into MVC, though. This means getting the repository working with a controller. For GetOrganized, we'll take the TodoController and have it retrieve information directly from the TodoRepository. This will finally rid us of the static list Todo.ThingsToBeDone.

To do that, we'll use an open source product called the Castle Windsor container that will help create controllers with their respective repositories. That will leave us in a good place to begin test-driving controllers by mocking out the repositories.

Once we have repositories integrated with controllers, we can simplify the way we validate models. We'll apply NHibernate's built-in validation framework to our MVC models. This makes validations easier to read and implement.

Finally, we'll take a look at NHProfiler, a commercial NHibernate profiler that can help you easily identify performance problems with your usage of NHibernate. It's worth knowing the basics of a profiler like NHProfiler to help spot performance problems early on.

Some of the code in this chapter was adapted from the open source project Sharp-Architecture.[1] Sharp-Architecture provides a highly testable architecture for an ASP.NET MVC application. Here it has helped with setting up the NHibernate session and handling the database transactions.

1. http://www.sharparchitecture.net/

The first thing we need to do to integrate NHibernate into MVC is to get the NHibernate session going.

9.1 Fixing the NHibernate Session Inside MVC

For NHibernate to work with MVC, we'll need a place to store the NHibernate ISession. For this job, we'll create a class called NHibernateSessionStorage located in the Persistence in the Web project right next to the NHibernateConfiguration class. It will be responsible for retrieving and opening the NHibernate ISession in the HttpContext for every web request that comes in.

Download integratingRepositoriesWithControllers/NHibernateSessionStorage.cs

```
Line 1  public class NHibernateSessionStorage
     -  {
     -    private const string CURRENT_SESSION_KEY =
     -      "nhibernate.current_session";
     5
     -    public static ISession RetrieveSession()
     -    {
     -      HttpContext context = HttpContext.Current;
     -      if (!context.Items.Contains(CURRENT_SESSION_KEY)) OpenCurrent();
    10      var session = context.Items[CURRENT_SESSION_KEY] as ISession;
     -      return session;
     -    }
     -
     -    private static void OpenCurrent()
    15    {
     -      ISession session = NHibernateConfiguration.CreateAndOpenSession();
     -      HttpContext context = HttpContext.Current;
     -      context.Items[CURRENT_SESSION_KEY] = session;
     -    }
    20
     -    public static void DisposeCurrent()
     -    {
     -      if (!HttpContext.Current.Items.Contains(CURRENT_SESSION_KEY))
     -        return;
    25      ISession session = RetrieveSession();
     -      if (session != null && session.IsOpen)
     -          session.Close();
     -      HttpContext context = HttpContext.Current;
     -      context.Items.Remove(CURRENT_SESSION_KEY);
    30    }
     -  }
```

The public method RetrieveSession() returns an NHibernate ISession. On line 8, we obtain the current HttpContext, which is used as the storage device. We then check to see whether an ISession exists in the collection of Items under the key CURRENT_SESSION_KEY. This makes the method RetrieveSession() safe to call multiple times by either opening a new ISession on line 14 or returning the one that has already been opened for this web request. If the request has no ISession, then OpenSession() is called and asks NHibernateConfiguration to create a new one. It then stores the ISession back into the HttpContext.

The other public method, DisposeCurrent(), closes and releases the NHibernate ISession from the HttpContext. On line 24, we make sure that an ISession exists in the HttpContext, because only requests that use NHibernate will have one. If there is no ISession, then we don't want to call RetrieveSession(), because that opens a new one.

Now we'll put the NHibernateSessionStorage and NHibernateConfiguration to work. We'll need to modify Global.asax.cs to initialize NHibernate and make sure every web request has an ISession. Also, Global.asax.cs will be responsible for safely disposing of the NHibernate ISession once the web request is finished.

Download integratingRepositoriesWithControllers/Global.asax.cs

```
public class MvcApplication : HttpApplication
{
  private readonly object lockObject = new object();
  private bool wasNHibernateInitialized;

  private void Application_BeginRequest(object sender, EventArgs e)
  {
    InitializeNHibernate();
  }

  private void Application_EndRequest(object sender, EventArgs e)
  {
    NHibernateSessionStorage.DisposeCurrent();
  }

  private void InitializeNHibernate()
  {
    if (!wasNHibernateInitialized)
    {
      lock (lockObject)
      {
        if (!wasNHibernateInitialized)
        {
          NHibernateConfiguration.Init(
              MsSqlConfiguration.MsSql2005.
```

```
               ConnectionString(builder => builder.
               FromConnectionStringWithKey("ApplicationServices")),
               UpdateDatabase());

30           wasNHibernateInitialized = true;
          }
        }
      }
    }

35  private Action<Configuration> UpdateDatabase()
    {
        return config => new SchemaUpdate(config).Execute(false, true);
    }
40 }
```

Every web request that comes into MVC triggers the method Application_BeginRequest(). Alternatively, you can create an HttpModule that runs the same code and wire it up in Web.config. Here we'll just modify the ASP.NET life-cycle events directly in Global.asax.cs for ease of readability.

Application_BeginRequest() checks to see whether NHibernate has been initialized. It *never* calls RetrieveSession() as you might expect. This is because there might be some controllers that don't require database connectivity and therefore never use a session. Instead, an ISession is retrieved only when a repository needs one. This will be covered a little later in Section 9.2, *Using Factory Methods in Castle Windsor to Retrieve Sessions*, on page 208.

The private method InitializeNHibernate() calls NHibernateConfiguration in a thread-safe way. It uses the two private members, lockObject and wasNHibernateInitialized, to ensure thread safety. On line 18, we quickly check to see whether NHibernate has been initialized already, since we want to do this only once. Next on line 20, we use the keyword **lock** to instruct the program to get an exclusive lock on lockObject. This prevents any other thread (incoming web request) from running the rest of this code while the current thread does. To be extra safe, we'll do a second check on the wasNHibernateInitialized on line 22 to make sure that milliseconds before we got the lock another thread didn't already initialize the NHibernate.

Once we're sure about the first thread to initialize NHibernate, we call NHibernateConfiguration to do the job. We pass it the connection string from the Web.Config on line 27. This is the same connection string we set in Section 5.2, *Set Up SQL Server for the Membership Provider*, on page 109. Also, the database schema is set to be upgraded on line 28 by calling Execute(bool generateSQLscript, bool performUpgradeOnDatabase).

Just as we want to ensure we create only one NHibernate session per web request, we also want to safely dispose them once the request is complete. The method Application_EndRequest() is called by ASP.NET after the request is completed. This is where we call NHibernateSessionStorage's DisposeCurrent() method to clean up this resource.

This is all the setup we need to have one NHibernate ISession per request. We can now move on to initializing our repositories and ISessions using an Inversion of Control (IoC) container.

9.2 Using Inversion of Control with the IControllerFactory

As your web application gets larger and more complex, there tends to be a lot more duplication of code. This can happen when you have a team of developers who aren't on the same page about how you do something like creating a repository. For example, the easiest way to create a new repository is to directly invoke the **new** keyword. Easiest isn't always the smartest. After a while, you'll catch yourself repeating this line of code over and over again.

One way to avoid this repetition is to create a RepositoryFactory class. The factory will give you a particular repository in a state that is configured and ready to use. Now all the configuration of the particular repository is in one place. So, the RepositoryFactory helps reduce the duplication and allows developers on teams to create objects in a consistent manner. They do have a downside, however. If repositories start to have their own dependencies on other objects, then you'll end up making the RepositoryFactory more complicated than it needs to be.

Luckily, these headaches of dependency and object creation can be solved by using an Inversion of Control container. An IoC container manages and resolves dependencies between objects. They also centralize object creation, similar to the RepositoryFactory, except they do this for all classes of objects in your application.

For our IoC container, we'll be using the open source Castle Windsor container. There are many IoC containers in the .NET space, other than

Castle Windsor. StructureMap,[2] Ninject,[3] Unity,[4] and Spring.NET,[5] all perform the same job in roughly the same way. We're going to use the Castle Windsor container because it is among the most popular and has a lot of documentation for working with both MVC and NHibernate.

Now let's hook the Castle Windsor container into MVC.

Treat Your Objects like Royalty at Castle Windsor

Back in Section 5.1, *IControllerFactory: Where Controllers Are Born*, on page 106, you saw that the IControllerFactory is the entry interface for the creation of all controllers. Naturally, this is the most logical place to create and inject the repositories into the controllers.

First, download Castle Windsor 2.0,[6] and add the following DLLs to your Lib folder:

- Castle.Core.dll(you already have this from FluentNHibernate)
- Castle.DynamicProxy2.dll (you already have this from FluentNHibernate)
- Castle.MicroKernel.dll
- Castle.Windsor.dll

You'll need to add these references to your *Web* project to use the container.

Creating objects in Castle Windsor is fairly straightforward. You first need to register the type of classes with the Castle Windsor container and then ask the container for an object of that type:

Download integratingRepositoriesWithControllers/WindsorSample.cs

```
IWindsorContainer container = new WindsorContainer();
container.Register<Foo>();
Foo newFoo = container.Resolve<Foo>();
```

Here we create a **new** Castle Windsor container, register a Foo, and then obtain a new Foo. You use the Register() method to register a type and call Resolve() to get a new object. The default behavior of Castle Windsor is to return the exact object that was registered. In Castle Windsor

2. http://structuremap.sourceforge.net/Default.htm
3. http://ninject.org
4. http://www.microsoft.com/downloads/details.aspx?displaylang=en\&FamilyID=ab3f2168-fea1-4fc2-b40c-7867d99d4b6a
5. http://www.springframework.net
6. http://sourceforge.net/projects/castleproject/files/InversionOfControl/2.0/Castle-Windsor-2.0.zip/download

terminology, this is known as having a LifestyleType of Singleton, meaning we only ever have one of these objects in the system. Most of the time we want a new object, so we'll set the LifestyleType to Transient. You can register an object with a specific lifestyle like this:

Download integratingRepositoriesWithControllers/WindsorSample.cs

```
container.Register<Foo>(LifestyleType.Trasient);
```

That's how to use Castle Windsor in its simplest way. Now it's time to use Castle Windsor to register both controllers and repositories. To get those controllers out, we're going to need to change the DefaultController-Factory to do that. Luckily, MVCContrib already has written the code to take care of this for us.

MVCContrib Extras (not part of the core MVCContrib.dll download) has two classes related to this function: WindsorControllerFactory and WindsorExtensions. Extensions adds the method RegisterControllers() to the IWindsorContainer, allowing us to register all the controllers in our Web project.

To register a Windsor container as our dependency injection framework, we need to add a reference to MvcContrib.dll in our *Web* project. Unfortunately, the file containing the two mentioned classes that are in MvcContrib.Castle.dll uses an old version of Windsor (1.0) and we're using 2.0. Not to worry—it's open source, so we can just take these classes into our codebase.

Let's start with how the factory code works:

Download integratingRepositoriesWithControllers/WindsorControllerFactory.cs

```
public class WindsorControllerFactory : DefaultControllerFactory
{
  private readonly IWindsorContainer container;

  public WindsorControllerFactory(IWindsorContainer container)
  {
    this.container = container;
  }

  protected override IController GetControllerInstance(
    RequestContext context, Type controllerType)
  {
    if(controllerType == null)
    {
      throw new HttpException(404,
        string.Format("The controller for path '{0}' " +
        "could not be found or it does not implement IController.",
        context.HttpContext.Request.Path));
    }

    return (IController) container.Resolve(controllerType);
  }
}
```

WindsorControllerFactory extends the DefaultControllerFactory and overrides the GetControllerInstance(). This checks for a **null** type and then uses an overload of the Resolve(Type typeToReturn) method to return the appropriate controller.

Now that we know what an IControllerFactory implementation looks like, we need to replace the DefaultControllerFactory with the new WindsorControllerFactory in the Global.asax.cs.

Download integratingRepositoriesWithControllers/Global.asax.cs

```
public class MvcApplication : HttpApplication
{
  //omit NHibernateConfiguration code...

  protected void Application_Start()
  {
    SetupWindsorContainer();
    RegisterRoutes(RouteTable.Routes);
  }

  private void SetupWindsorContainer()
  {
    IWindsorContainer container = new WindsorContainer();

    RegisterControllers(container);
    RegisterNHibernateSessionFactory(container);
    RegisterRepositories(container);
  }

  private void RegisterControllers(IWindsorContainer container)
  {
    ControllerBuilder.Current.
      SetControllerFactory(new WindsorControllerFactory(container));
    container.RegisterControllers(typeof (HomeController).Assembly);
  }
}
```

We only want to register Castle Windsor once, so the Application_Start() method is the right place for the job. It is called once when the application starts up when the first web request arrives. In this method, we'll call SetupWindsorContainer(), which contains three **private** methods that will set up the container.

We are registering the WindsorControllerFactory in the RegisterControllers(IWindsorContainer container) method. We'll talk about the RegisterNHibernateSessionFactory() and RegisterRepositories() later in Section 9.2, *Using Factory Methods in Castle Windsor to Retrieve Sessions*, on the next page. To register the controllers, we access the ControllerBuilder on line 22. This object lets us assign the WindsorControllerFactory with the Castle Windsor container as the new IControllerFactory. On line 24, we use the WindsorExtensions class's extension method to register all the controllers in the same assembly as the HomeController. Let's take a peek at what WindsorExtensions looks like:

Download integratingRepositoriesWithControllers/WindsorExtensions.cs

```
public static class WindsorExtensions
{
  public static IWindsorContainer RegisterControllers(
    this IWindsorContainer container, params Assembly[] assemblies)
  {
    foreach (Assembly assembly in assemblies)
    {
      container.RegisterControllers(assembly.GetExportedTypes());
    }
    return container;
  }

  private static void RegisterControllers(
    this IWindsorContainer container,params Type[] controllerTypes)
  {
    foreach (Type type in controllerTypes)
    {
      if (ControllerExtensions.IsController(type))
      {
        container.AddComponentLifeStyle(type.FullName.ToLower(),
          type, LifestyleType.Transient);
      }
        }
  }
}
```

This class has two methods that use a bit of .NET reflection to add controllers to the container. The first method takes a list of assemblies and then calls the second method to do the registration itself. We use the LifestyleType.Transient to register the controller by name so that we always get a new controller instantiated for every request.

This takes care of registering controllers with Castle Windsor, but we also need to register other components like our repositories and the NHibernate ISessions.

Using Factory Methods in Castle Windsor to Retrieve Sessions

A **static** method that returns a newly created object is called a *factory method*. NHibernateSessionStorage has a nontraditional factory method called RetrieveSession(). It is nontraditional in the sense that the method returns the same object every time it is created, whereas a true factory will produce new objects every time. In this case, we want the same ISession to be used for the lifetime of a single web request. This is because NHibernate ISessions are expensive to create.

Castle Windsor has a special way to register factory methods. This will provide a way for the repositories we register to obtain an ISession.

Download integratingRepositoriesWithControllers/Global.asax.cs

```
Line 1  private void RegisterNHibernateSessionFactory(
     2    IWindsorContainer container)
     3  {
     4    container.AddFacility<FactorySupportFacility>();
     5    container.Register(Component.For<ISession>().
     6      UsingFactoryMethod(() =>
     7      NHibernateSessionStorage.RetrieveSession()).
     8      LifeStyle.Is(LifestyleType.Transient));
     9  }
```

A *facility* is an extension point of the Castle Windsor framework. Here we are using the FactorySupportFacility on line 4. The facility gives providers a way to register factory methods, which we do on line 5, for the ISession type. We call the method UsingFactoryMethod() and pass it a lambda expression on line 6 to specify that the source of the ISession is on the factory method RetrieveSession(). Finally, we instruct Castle Windsor that the ISession is Transient on line 8 so that we get a new one every time one is requested.

With ISessions registered with the factory method, the last step is to register the repositories themselves:

Download integratingRepositoriesWithControllers/Global.asax.cs

```
private void RegisterRepositories(IWindsorContainer container)
{
  IEnumerable<Type> repositories = Assembly.GetExecutingAssembly().
    GetTypes().Where(IsRepository);

  foreach (Type repository in repositories)
  {
    container.AddComponentLifeStyle(repository.Name, repository,
      LifestyleType.Transient);
  }
}
```

```
 -    private bool IsRepository(Type type)
 -    {
15      return type.Namespace != null && type.IsClass && !type.IsAbstract &&
 -        type.Namespace.Contains("GetOrganized.Web.Persistence.Repositories");
 -    }
```

On line 4, we obtain a list of repositories from the executing assembly (GetOrganized.dll). We restrict the types of classes we get back using the LINQ method Where() with the expression IsRepository() on line 15. This method filters back classes that are concrete (not **abstract** or an **interface**) and in the **namespace** GetOrganized.Persistence.Repositories. For each repository, we register it as transient on line 9 so every request has its own copy.

We have completed registration of the controllers, the repositories, and the NHibernate ISession. Our Castle Windsor container is now ready to serve a web request near you. We can now test-drive our controllers using our newly registered repositories.

9.3 Injecting Repositories into Controllers

All of this registration is the background work to replacing **static** lists with repositories. We learned how to test-drive the repository back in Section 8.4, *Creating and Reading Records*, on page 190. It's time now to test-drive controllers that contain one or more repositories. We do this by mocking out the repository using Rhino Mocks.

Once you become comfortable with mocking the repository, this will become the preferred order for test-driving your code. By starting with controller tests, you get a better idea of what data you will need based on the action it performs.

For example, while test-driving the controller action ProcessOrder(), you uncover that you need to modify both Order and Customer models. If you had started test-driving the OrderRepository first, you might not have seen the need to modify Customer until later. By understanding your coding requirements earlier, you will reduce this kind of rework.

Let's put this into practice. We're going modify the TodoController, which we last left off in Section 5.3, *Test-Driving Authorization*, on page 118. For this test, we're going to work on the Index() action. Let's test-drive this by mocking the TodoRepository.

Download integratingRepositoriesWithControllers/TodoControllerTest.cs

```
[TestFixture]
public class TodoControllerTest
```

```
 -  {
 -    private TestControllerBuilder builder;
 5    private TodoController todoController;
 -    private MockRepository mocks;
 -    private ISession session;
 -    private TodoRepository todoRepository;
 -
10    [SetUp]
 -    public void setup()
 -    {
 -      mocks = new MockRepository();
 -      builder = new TestControllerBuilder();
15          session = mocks.DynamicMock<ISession>();
 -      todoRepository = mocks.StrictMock<TodoRepository>(session);
 -      todoController = new TodoController(todoRepository);
 -      builder.InitializeController(todoController);
 -
20      var routes = RouteTable.Routes;
 -      routes.Clear();
 -      RouteDefinitions.AddRoutes(routes);
 -    }
 -
25    [Test]
 -    public void Should_Display_Todo_List_And_Logged_In_Users_Name()
 -    {
 -      const string userName = "Jonathan";
 -      var todoList = new List<Todo>
30        { new Todo { Title = "Refactor to NHibernate" } };
 -
 -      builder.HttpContext.User =
 -        new GenericPrincipal(new GenericIdentity(userName), null);
 -
35      Expect.Call(todoRepository.GetAll()).Return(todoList);
 -      mocks.ReplayAll();
 -
 -      var viewData = todoController.Index().
 -        AssertViewRendered().ViewData;
40
 -      Assert.AreEqual(todoList, viewData.Model);
 -
 -      Assert.AreEqual(userName, viewData["UserName"]);
 -      mocks.VerifyAll();
45    }
 -  }
```

To add a mock object, we set it up in the Setup() portion so that it is available to all tests in the fixture. On line 16, we use StrictMock<Todo-Repository>() to create the mock object. We then inject that mock into the TodoController constructor on line 17.

The test itself will use the Expect object, which is part of Rhino Mocks, which we already downloaded and installed in Section 5.3, *Using MVC-*

Contrib's TestControllerBuilder to Test Controllers, on page 119. This object is used to add expected behavior to the mock repository. On line 35, we set the expectation that TodoRepository's method GetAll() will be called and will return the todoList. The expectation needs to be replayed in the mocks collection itself, so on line 36 we use the method ReplayAll(). Alternatively, you can call the mock object explicitly using Replay(object mockObject).

Rhino Mocks objects need to be replayed for them to emulate the desired behavior. Other mock frameworks, like NMock,[7] inherently do this for you but at the cost of having to hard-code the names of the methods you want to mock in a string.

Mock objects will cause the test to fail only if they are improperly or never called. For this to happen, we need a verification step. All mock frameworks (including Rhino Mocks and NMock) have this built into them. After our normal assertions, we call VerifyAll() at the end of the test on line 44. Again, there is a second flavor of Verify(object mockObject) that verifies only a single mockObject. This test won't compile until we create a new constructor for TodoController that accepts a TodoRepository. Let's get this test to pass:

Download integratingRepositoriesWithControllers/TodoController.cs

```
[Authorize]
public class TodoController : Controller
{
  private readonly TodoRepository repository;

  public TodoController(TodoRepository repository)
  {
    this.repository = repository;
  }

  //
  // GET: /Todo/

  public ActionResult Index()
  {
    ViewData["UserName"] = User.Identity.Name;

    ViewData.Model = repository.GetAll();

    return View();
  }
}
```

7. http://nmock.org

First we add the new constructor on line 6. The repository is set to a **private readonly** member to ensure that it is never modified after construction. In the Index() action itself, we simple remove the call to the **static** list. Instead of using Todo.ThingsToBeDone, we now call todoRepository's GetAll() method on line 18. This makes the test pass. We've finally removed our **static** list and replaced it with a more testable repository.

The rest of the GetOrganized solution needs to be converted in a similar way. Any time you have a call to the static list, it will be replaced with a call to the repository. The full conversion is available in the downloadable code, available in the GetOrganizedFinal/ folder.

Another important aspect to ensuring our controllers work with repositories is to have them manage NHibernate transactions.

9.4 Creating a Custom Action Filter: The (Transaction) Attribute

Transactions are a programmatic boundary that marks the start and end of one or more database operations. They are an "all or none" guarantee that commits all the operations or rolls them all back if something goes wrong.

Most operations that update or save models ought be transactional, such as if there are multiple models being modified but one of those updates goes wrong and we don't want any of those changes saved. Otherwise, it'd be hard to know which models were modified and which failed. Let's say it's because of invalid information in only one of the models. In this case, you don't want the system to have updated and saved half or three quarters of the models.

This is where a transaction can save you from worrying about these scenarios. They save you from the nightmare of inconsistent data brought about by the application. Trust me, you don't want to explain to your boss that it was your code that created database soup.

What transactions don't do is ensure against stale data. For example, if one person is on the edit screen and another person clicks the save button first, then transactions will not save you from this headache. One technique is to perform a checksum on the data and before saving the data check to see whether the data is consistent with what it was before you are making the change. Alternatively, you can compare the timestamps on the records.

Those who know transactions from ADO.NET will be familiar with creating a **using** block around a System.Transactions.Transaction object. This